REASONING WITH WORLDS AND TRUTH MAINTENANCE IN A KNOWLEDGE-BASED PROGRAMMING ENVIRONMENT

In traditional knowledge-based system development environments, the fundamental representational building blocks are mechanisms such as frames, rules, and attached procedures. The KEE system has been extended to include both a context (worlds) system and a truth maintenance system.

ROBERT E. FILMAN

Broadly speaking, computers are information transducers: They read data, manipulate the data in some computational process, and display the results. The mapping we create between the input data, the manipulation and output data, and the external world is part of what makes computers valuable. The payroll program that takes wage rates and hours worked, multiplies and figures deductions, and prints the values for net paychecks is useful because the wage rates and hours worked, computations, and net checks correspond to the real hours, rates, etc., of the company's workers. Traditionally, computers have been limited to modeling only those parts of the world where regularity dominates exception. That is, because workers are paid the product of their wage rate and hours worked—less deductions computed by simple formulas and table lookups—it is straightforward to write a computer program that computes paychecks. Because a company has many workers who it repeatedly pays by the same algorithm, it is worthwhile for it to have such a program written. To the extent that the underlying world is more complex, the program required for the modeling becomes more complex, and takes (perhaps exponentially) more skill to design, more time to write, more effort to debug, and more devotion to maintain. A pay system based, for example, on the complexity of task performed, the external demand for the objects produced, predictions about future economic conditions, and the artistic quality of the work is beyond the reach of conventional programming technology. Dealing with that degree of complexity requires more sophisticated systems than are currently available.

The goal of knowledge-based systems (KBS) technology is to greatly expand the horizon of "reasonable-to-build" applications. The use of KBS technology simplifies modeling a large class of complex situations involving symbolic reasoning and eases the task of stating complicated things about irregular domains. Nevertheless, even this expressiveness would not be useful without the tools to make it accessible—inspection and modification mechanisms to reveal the state of the model, and input and display mechanisms to easily translate between computer and human-understandable forms. Toward this end, several *KBS development environments* have been developed, both in research institutions and commercially. These environments use technologies such as pattern-action rules (Emycin [26], OPS-5 [2], ART® [27], S-1® [10]), frames (Units [23], KL-ONE [1]), variants of procedural attachment (i.e., daemons and object-oriented programming) (Smalltalk [9], Flavors [25]), and integrations of the above (Loops [24], KEE® [6]). Such environments provide not only

This research was supported by the Defense Advanced Research Project Agency under Contract F30602-85-C-0065. The views and conclusions reported here are those of the author and should not be construed as representing the official position or policy of DARPA, the U.S. government, or IntelliCorp.

ART is a trademark of Inference Corporation.
S-1 is a trademark of TeKnowledge.
KEE is a trademark of IntelliCorp.

the internal representational structures of their chosen paradigm, but also interface facilities that understand and can manipulate these structures.

Recently, we have extended the KEE environment to include both a *truth maintenance system* (*TMS*) (based on de Kleer's work on the assumption-based truth maintenance system (ATMS) [3]) and a context or *worlds* system [19]. We call these extensions *KEEworlds*®. A world represents a set of related facts—for example, a situation, a simulation checkpoint, a belief set, or a hypothetical state of a problem solver. A world is characterized by a set of *assumptions*. The TMS remembers the assumptions on which each deduced fact is based. A world sees a deduced fact if and only if the world's assumptions are a superset of the assumptions that support that deduction.

The integration of a conventional object-oriented representation environment with worlds and truth maintenance is a novel combination. It required modifying the system's internal representation structures, and constructing a new rule system to manipulate world and ATMS entities. Similarly, discovering how best to exploit these new facilities requires further research and experimentation. This article presents some of our early experiments with the new system.

The primary activity of a KEE system user is first constructing a *model* of an underlying domain and then building one or more reasoning components that manipulate that model. Thus, KEE is a tool that enables *model-based reasoning*. In this article we develop several examples of reasoning with KEEworlds, all centered around a common domain of scheduling shipments. As truth maintenance may be unfamiliar to some readers, we provide a short overview and history of truth maintenance in the accompanying sidebar.

AN OBJECT-CENTERED DOMAIN DESCRIPTION

We illustrate our discussion of knowledge representation and reasoning with examples from the problem of building tools to aid the dispatcher of the hypothetical Big Giant Trucking Company. Big Giant serves 24 *cities* in Indiana and Illinois (Figure 1 shows an area we call `Mid.Continent`) moving *shipments* of various materials over particular *highways* in certain *trucks* driven by specified *drivers*.

The dispatcher wants to devise a *schedule*—a collec-

KEEworlds is a trademark of IntelliCorp.

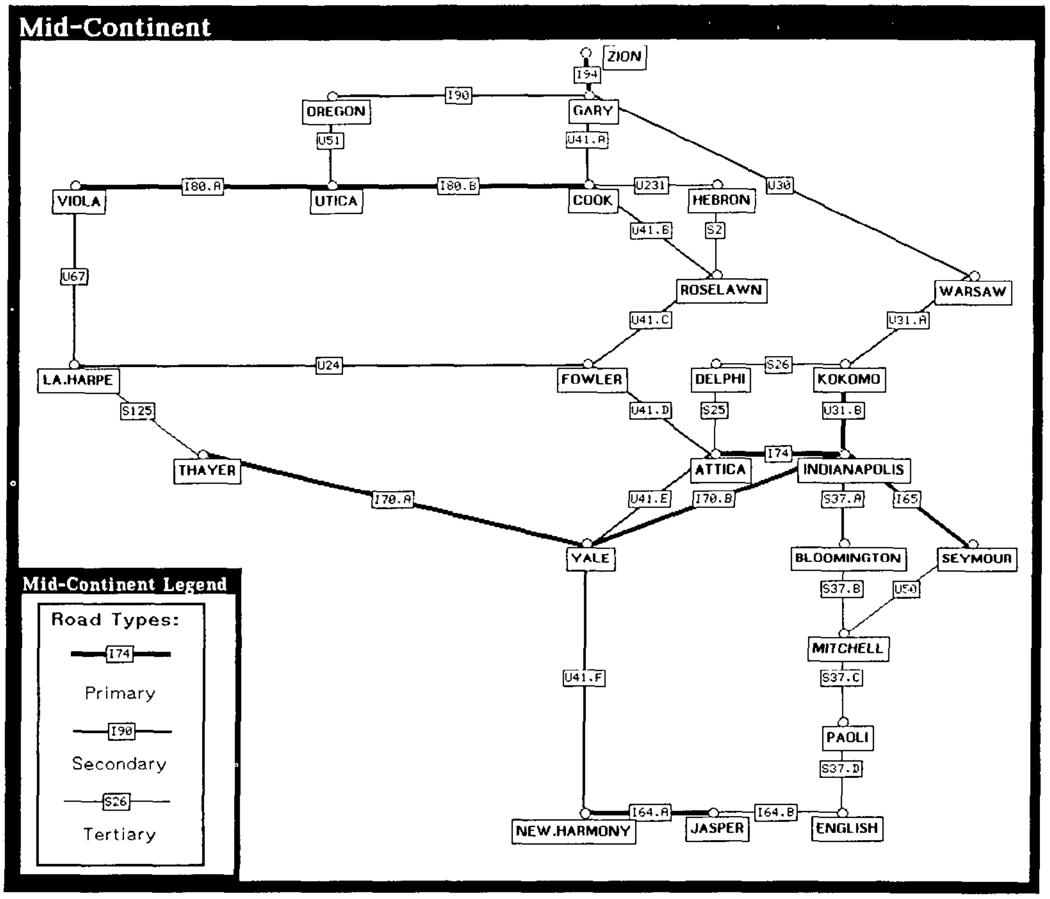


FIGURE 1. `Mid.Continent`

An Overview of Truth Maintenance

In general, reasoning is the process of deriving new knowledge from old. If the underlying knowledge never changes, if we never explore hypothetical spaces, and if our knowledge is free of internal contradictions, the accumulation of knowledge is straightforward: We just add the results of our reasoning to our pile of knowledge. Unfortunately, few problems are so simple. We usually find ourselves reasoning under a set of assumptions that may be withdrawn or changed. Often the entire reasoning process is focused on identifying preferred assumption sets. Ideally, when the assumptions change, we would like to withdraw those conclusions that are no longer valid, retaining those that are still true. This requires attaching to derived facts *justifications* or *dependencies*, that is, reasons for belief in these facts.

Historically, the need for dependencies first arose in the context of the *frame problem* [16]—the problem of determining what has not changed over an event or series of events. For example, we imagine situation S, where a monkey is in a room with a red box located at position $\langle x, y \rangle$. The action of the monkey, A, of *pushing the box to* $\langle x', y' \rangle$ creates a new situation, S'. How is our computer system to know that the color of the box is still red in S'? It is not the case that the color of the box is constant over all actions. Instead, if A had been the action *painting the box green*, then the color of the box would be different in S'. Our system must somehow incorporate the knowledge that the action of moving an object does not change its color.

Systems such as STRIPS [7] and PLANNER [11] approached the frame problem by associating with each action lists of facts that were added by the action and facts that were deleted by the action. The problem with this approach is that to be correct the operators that changed system state had to modify all facts that had been derived on the basis of the now-to-be-deleted facts. That is, if in state S, above, we had concluded that the box at $\langle x, y \rangle$ was under a bunch of bananas, and that action A was moving the box, we needed to withdraw this conclusion in S'.

One of the first systems to associate dependencies with derivations was Stallman and Sussman's system for circuit analysis, EL [22]. Their goal was to find those faulty assumptions responsible for producing contradictions. They introduced the idea of *dependency-directed backtracking*. Traditionally, many systems have relied on *chronological backtracking*, that is, considering all the possibilities for the most recent choice before revising any earlier decision. Chronological backtracking has the advantage that it is simple to implement with a stack.

We illustrate the disadvantages of chronological backtracking with a variant on the monkey-and-bananas problem. In the traditional monkey-and-bananas problem, a monkey is in a room, and the room has a bunch of bananas hanging from the ceiling and a box on the floor. The monkey wants to get (and then eat) the bananas. To achieve this goal, the monkey must push the box under the bananas, climb the box, and grab the bananas. In our problem, our monkey comes into a room with several boxes and several bunches of hanging bananas. The monkey's goal is once again to obtain a comestible bunch. The monkey proceeds to select a bunch of bananas, select a box, push the box under the bananas, climb, and grab. But lo—the bananas are sour. The monkey has a failure. Chronologically backtracking, the monkey reconsiders the last decision, the box selection. So the monkey picks another box, climbs down, pushes the first box away, pushes the new box under the same bunch of bananas, and so forth. Only after the monkey has exhausted all the boxes in the room does the chronologically backtracking monkey reconsider the choice of which bunch of bananas to pursue. A monkey using dependency-directed backtracking would notice that the sourness of the bananas depended only on the bunch choice (independent of the box choice), and would revise that choice instead. That is, in dependency-directed backtracking, the choice to be revised is not simply the last choice made, but a choice that contributed to the failure. To be able to do this, we must keep the dependencies of derivations. (There are, of course, other salutary effects from retaining dependencies for conclusions. The most important of these is that we are keeping the information required to explain the derivation and validity of those conclusions.)

Doyle [5] and, independently, London [12] were the first to recognize that the facilities for recording dependencies, dependency-directed backtracking, and "currently believing" particular assumptions could be incorporated into a system independent of an overarching reasoning mechanism. Doyle called his system a *truth maintenance system* or *TMS* (the term has stuck, though he currently favors the phrase *reason maintenance system*). In addition to dependencies, Doyle's system incorporated the idea that particular assumptions could be *in* (currently believed) or *out* (not currently believed); a particular derivation would be valid, for example, if assumptions $\mathcal{X}$ and $\mathcal{Y}$ were *in*, but $\mathcal{Z}$, *out*. In-ness and out-ness enable both modeling varying "current worlds" (worlds being assignments of in and out to assumptions), and basing beliefs on the out-ness of facts (in the spirit of Planner's THNOT (not found) [11]). Issues in the implementation of TMSs arise in the algorithms for revising the beliefs of the system when assumptions go in and out. In general, algorithmic and semantic difficulties can ensue when revising beliefs that have (circularly) come to support themselves.

TMSs have been a fertile field for artificial intelligence (AI) research, for example, the work of de Kleer et al. [4], McAllester [14], McDermott [17], and Martins and Shapiro [13]. In our work, we have been extending de Kleer's assumption-based truth maintenance system (ATMS) [3] to include contextual mechanisms (worlds), nonmonotonicity (assumption retraction), and integration with an underlying frame system.[1]

[1] In the first two respects, our system appears similar in behavior to the Viewpoints™ facility of ART [27]. As little has been published about the algorithms of that system, however, it is difficult to make detailed comparisons.

™ Viewpoints is a trademark of Inference Corporation.

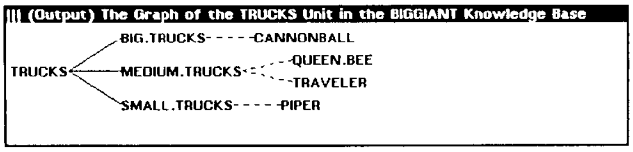


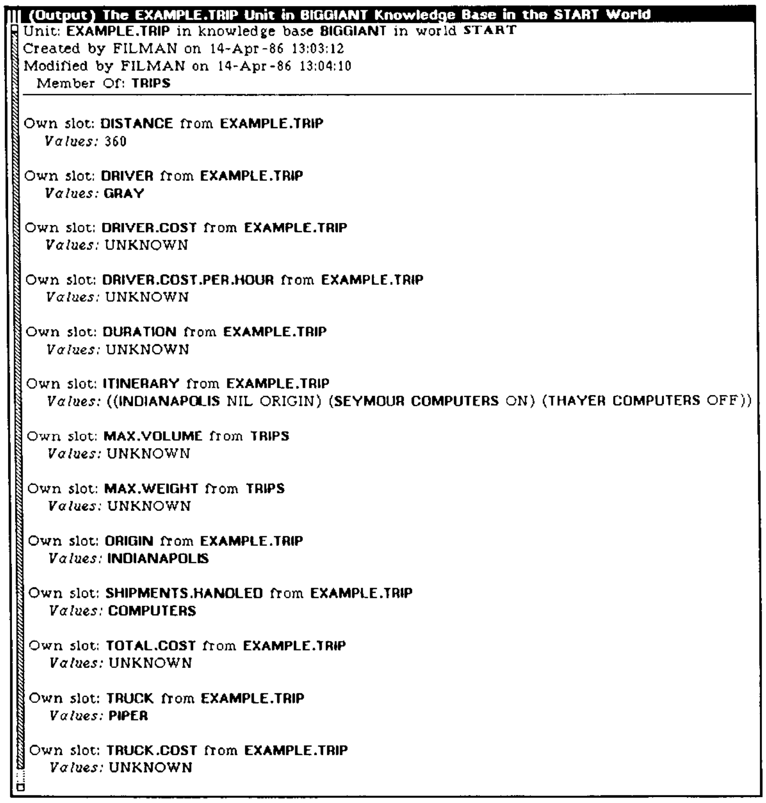
FIGURE 2. The `trucks` Hierarchy

tion of *trips* (an assignment of a truck and a driver to a particular itinerary) such that all shipments are picked up at their origins and delivered to their destinations. The dispatcher's job is complicated by the fact that he or she is working under a set of *constraints*—restrictions about what constitutes a legal schedule. These constraints range from common sense, for example, "The driver of a trip has to be in the same city as the truck" and "You can't put more on a truck than it can hold"; through the legalities of this particular domain, for example, "Union drivers can't drive more than 11 hours a day" and "You need the right kind of license to drive the bigger trucks"; and on to the absurdities that characterize so much of the real world, for example, "Driver White is wanted by the police in Illinois and can't be sent there." Ideally, the dispatcher would like to *optimize*—to construct a near-minimal cost schedule—but in a highly constrained situation, is usually lucky just to find a *feasible* (legal) schedule.

Although Big Giant is just a simple example developed to illustrate the points of this article, it shares with real problems a complex texture of regularity punctured by exceptions. Such a combination of regularity and exception characterizes domains most appropriate for knowledge-based techniques: The regularity of the domain enables us to actually build something, whereas the exceptions foil conventional programming technology.

We represent the objects of the domain as *units* (frames) and arrange these units in a class hierarchy; thus, the class of `trucks` has *subclasses* `big.trucks`, `medium.trucks`, and `small.trucks`. Individual trucks are *members* of these classes (Figure 2). For example, truck `Piper` is a member of the class of `small.trucks`. Both classes and members are represented as units; a given unit can represent both a class and a member of a class.

Relationships between objects and values are represented as *slots* in the units that represent the objects. There are two kinds of slots: *member* slots and *own* slots. An own slot expresses a relationship involving its unit as an individual. Thus, the statement that the value of the own slot `location` in unit `Piper` is `Gary` is the assertion that the location of Piper is Gary. Member slots occur only in class units. A member slot expresses a relationship involving the members of the class. The



```
||| (Output) The EXAMPLE.TRIP Unit in BIGGIANT Knowledge Base in the START World
Unit: EXAMPLE.TRIP in knowledge base BIGGIANT in world START
Created by FILMAN on 14-Apr-86 13:03:12
Modified by FILMAN on 14-Apr-86 13:04:10
  Member Of: TRIPS

Own slot: DISTANCE from EXAMPLE.TRIP
  Values: 360

Own slot: DRIVER from EXAMPLE.TRIP
  Values: GRAY

Own slot: DRIVER.COST from EXAMPLE.TRIP
  Values: UNKNOWN

Own slot: DRIVER.COST.PER.HOUR from EXAMPLE.TRIP
  Values: UNKNOWN

Own slot: DURATION from EXAMPLE.TRIP
  Values: UNKNOWN

Own slot: ITINERARY from EXAMPLE.TRIP
  Values: ((INDIANAPOLIS NIL ORIGIN) (SEYMOUR COMPUTERS ON) (THAYER COMPUTERS OFF))

Own slot: MAX.VOLUME from TRIPS
  Values: UNKNOWN

Own slot: MAX.WEIGHT from TRIPS
  Values: UNKNOWN

Own slot: ORIGIN from EXAMPLE.TRIP
  Values: INDIANAPOLIS

Own slot: SHIPMENTS.HANDLED from EXAMPLE.TRIP
  Values: COMPUTERS

Own slot: TOTAL.COST from EXAMPLE.TRIP
  Values: UNKNOWN

Own slot: TRUCK from EXAMPLE.TRIP
  Values: PIPER

Own slot: TRUCK.COST from EXAMPLE.TRIP
  Values: UNKNOWN
```

FIGURE 3. `Example.trip`

statement that the value of the member slot `location` in unit `trucks` is `Indianapolis` is (to a first approximation) the assertion that the location of any member of the class of `trucks` defaults to `Indianapolis`. *Facets* are annotations of slots to express additional information about that slot. The facets of member slots inherit along with the slot itself. Typical facets are *inheritance role* (the rule used to combine values from the unit's parents with the unit's local values), `valueclass` (type), and `min.cardinality` and `max.cardinality` (restrictions on the number of values a slot can have).

KEE provides a number of standard inheritance roles, such as *union* (the value in an inherited slot is to be the union of the local values and the inherited values), *override.values* (if there is a local value in the slot, it is the value of the slot; otherwise, the values inherited from some parent are used), and *method* (a mechanism for assembling functions from fragments, similar to the mix-ins of Flavors [25]). Users can describe additional inheritance roles of their own. Although KEE provides a variety of inheritance mechanisms (and allows user-defined extensions to this set), in these examples we use inheritance only to specify locally overridable default values.

Valueclass information is used to deduce type violations, to determine the semantic classes for values, to coerce ambiguous notation to the appropriate data type, and to organize particular interface mechanisms. *Cardinality* information is similarly used to detect *contradictions.* The behavior of the system on detecting a valueclass or cardinality violation (coercing the value to the new class, interrogating the user, or noticing a contradiction) is controlled by the setting of a global switch.

Since the task of the dispatcher is to create trips, it seems useful to reify the concept of a trip. We have a class of `trips`, and members of that class that are particular trips. Conceptually, the dispatcher believes that a trip has been composed when he or she has told a particular driver to drive around in a specific truck doing certain things. We represent these components of a trip as slots in the trip unit: `driver`, `truck`, and `itinerary`. We represent an itinerary as a sequence of *actions*, where each action is a triple: a city for the action to take place in, an object for the action (one of the shipments or nil), and the particular action to be taken (originating in that city, taking that shipment on or off the truck, or just visiting the city). This representation is strong enough to specify the route of a trip to the individual highway segment level, but flexible enough to allow us the more minimal specification of only the key actions of the trip, permitting the driver to take the usual (shortest) route between any two cities. From the specification of a driver, truck, and itinerary for a trip (and the weather), it is possible to derive other facts about a trip—for example, how long it takes and how much it costs. We also store such derived information in slots of the trip. Figure 3 shows the information in trip `Example.trip` partway through a problem-solving process.

WORLDS AND TRUTH MAINTENANCE

In general, problem solving is the discovery of some set of beliefs—be they the values of some variables, a complex data structure, or a collection of formulas in a theorem prover. In systems that search, different sets of beliefs are believed at different points in the problem-solving process. That is, we may start by believing X, conclude Y, and then switch context to believing Z. On the other hand, the entire search process proceeds against a background of a fixed set of facts—a model of the unchanging underlying world. Thus, if our task is to generate itineraries through the cities of `Mid.Continent`, we have at various points beliefs about partially assembled itineraries, and the costs and consequences of these itineraries. On the other hand, facts such as the connectivity network of cities and highways, the capacity of trucks, and the license class of drivers are constant throughout the problem-solving process. These *background facts* are true in every context. (Of course, if our problem solving included the possibility of improving drivers' licenses or adding routes to our territory, these would cease to be facts in the background.) The system represents (most) such background facts more economically, without incurring the space and time costs of truth maintenance.[2]

The ATMS is primarily concerned with those expressions that have different values in different contexts—the fodder of search. The ATMS records the justifications for beliefs, propagates justifications on the basis of new derivations, and ensures that exactly the appropriate derived facts are visible at any time. To accomplish this, the ATMS incorporates three basic concepts: *facts* (also called propositions or nodes), *assumptions,* and *justifications.*[3]

Formally, the ATMS manipulates *assumptions* and *propositions.* Each assumption corresponds to a primitive decision or choice. We use assumptions primarily either to hypothesize the existence of some context or to believe some particular fact. Each proposition has an associated *datum*, its content for the users of the ATMS. The datum, however, is not itself used by the ATMS operations. (Thus, each own-slot value and unstructured fact that has been noticed by the ATMS is the datum for its own unique proposition. Background facts economize by going without propositions.) We use the notation $\mathcal{P}$ to represent the proposition associated with fact P, and the notation $\ddot{\mathcal{P}}$ to represent the assumption of P—the choice of believing $\mathcal{P}$.

Propositions may be *justified* in terms of assumptions or other propositions. Justifying proposition $\mathcal{Z}$ by $\mathcal{X}$ and

[2] Background facts are not to be confused with defaults—defaults are a mechanism for easily expressing a bulk of information and exceptions for that information. In general, a particular default may or may not be true in the background.

[3] There are various ways that information can be stored in the KEE system: as the values of own and member slots and facets, in the inheritance links between units, and as unstructured facts (arbitrary data structures). The ATMS maintains the truth of only the values of own slots and unstructured facts. We have not implemented truth maintenance on statements such as `(Cannon-ball is in class large.trucks)` (class membership) and `(The transmission of all trucks is automatic)` (member-slot values). Some of our recent work has produced Opus, a KEE-like system where all facts are accessible to the ATMS [8].

$\mathcal{Y}$, $(X, \mathcal{Y} \vdash \mathcal{Z})$ is the assertion that, whenever X and $\mathcal{Y}$ are believed, $\mathcal{Z}$ is to be believed, too.

Viewing the justification of a proposition by a set of assumptions and propositions as a single proof step, we see that the justification structures for a particular proposition form proof trees for that proposition. An *environment* is the set of assumptions obtained by transversing such a justification structure back to a well-founded set of assumptions. The *label* of a proposition is the set of minimal environments that support that proposition. The label can be seen as a summary of the necessary assumptions required for believing the associated datum. The primary operation in the ATMS is the addition of a justification to a proposition. This causes the ATMS to update the labels of all affected propositions. That is, if we discover another set of assumptions that supports the belief in proposition $\mathcal{P}$, we consider for each proposition directly justified by $\mathcal{P}$ whether that set of assumptions is part of a new minimal support for it. This process ensures the label of a proposition always reflects every minimal set of assumptions that imply that proposition.

It is convenient to identify "assuming a datum in a context" with the proposition structure of that datum and with the datum itself. Data are distinguished from propositions because propositions include more information—proofs of the datum and summaries of the supporting assumptions of those proofs. Assumptions are distinguished from propositions because (1) propositions can acquire other justifications than just the decision to assume them, and (2) assumptions are used for the system's context mechanism. On the other hand, it is convenient in most situations to think of facts as identical with their propositions (and, occasionally, with the assumption of those facts).

Let us consider an example in greater detail. Suppose we come to justify the fact

`The truck.cost of some.trip is 452` ($\mathcal{A}$)

on the basis of the facts

`The truck of some.trip is Traveler` ($\mathcal{B}$)

and

`The itinerary of some.trip is....` ($\mathcal{C}$)

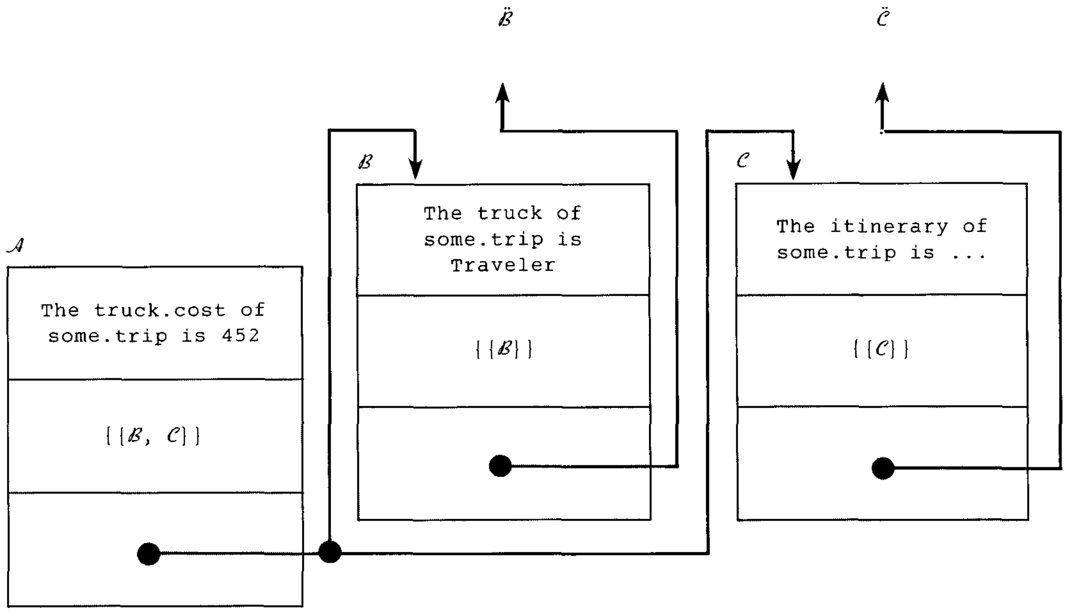
This belief might arise, for example, from the application of a rule about computing truck costs. In any context where we come to believe $\mathcal{B}$ and $\mathcal{C}$, we also believe $\mathcal{A}$. If our beliefs in $\mathcal{B}$ and $\mathcal{C}$ are based on assumptions $\ddot{\mathcal{B}}$ and $\ddot{\mathcal{C}}$, then the structure for the proposition whose datum is $\mathcal{A}$ points to the justification structures $\mathcal{B}$ and $\mathcal{C}$, and the label, $\{\{\ddot{\mathcal{B}}, \bar{\mathcal{C}}\}\}$ (Figure 4).

We might also come to justify $\mathcal{A}$ on the basis of facts $\mathcal{C}$ and

`The truck of some.trip is Queen.Bee,` ($\mathcal{D}$)

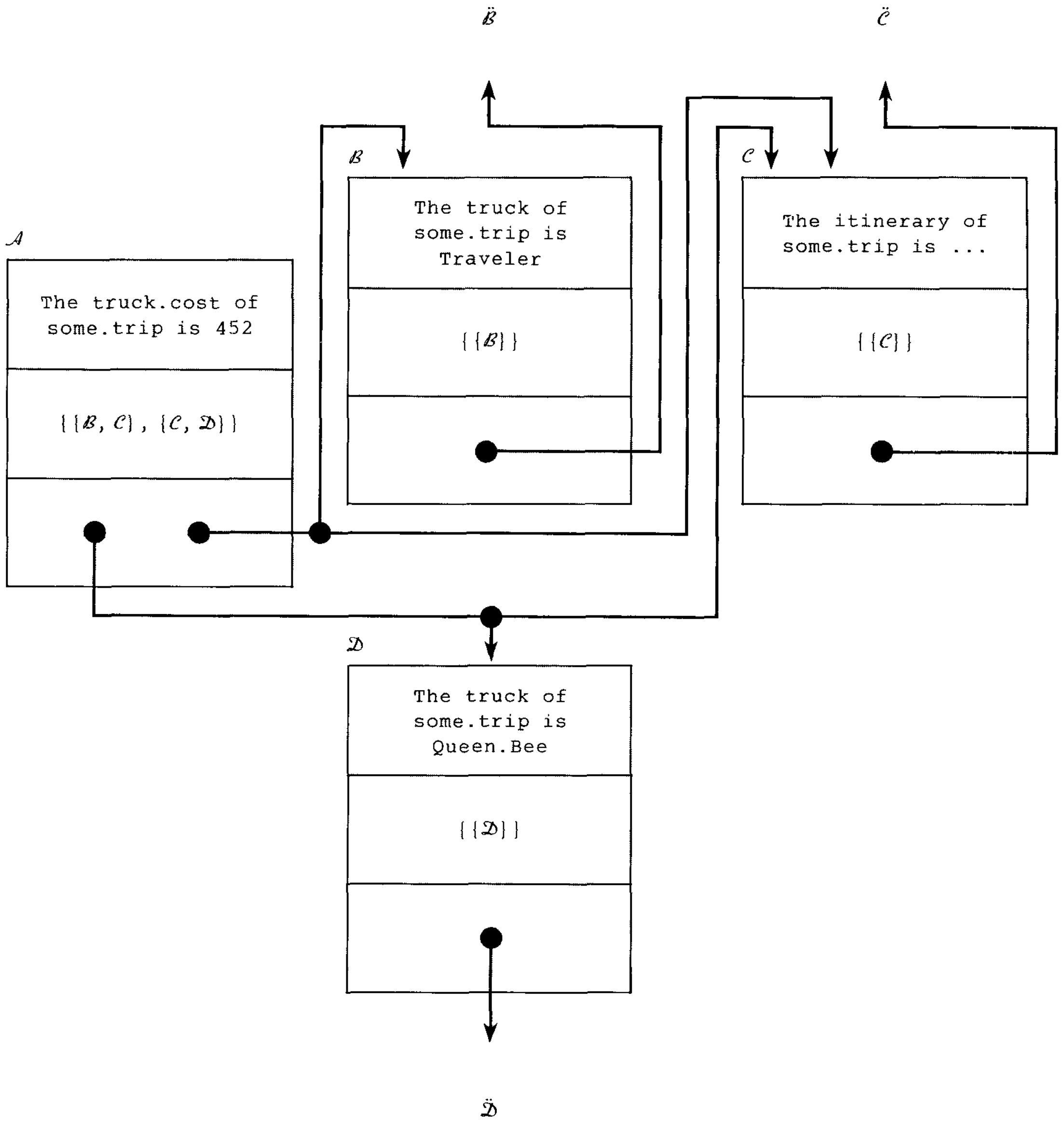
$\mathcal{D}$ similarly supported by assumption $\ddot{\mathcal{D}}$. The label of $\mathcal{A}$ would then be $\{\{\ddot{\mathcal{B}}, \ddot{\mathcal{C}}\}, \{\ddot{\mathcal{C}}, \ddot{\mathcal{D}}\}\}$. Our justification structure grows to that of Figure 5. If we discover a justification of $\mathcal{A}$ that traces back to assumptions $\ddot{\mathcal{B}}$, $\ddot{\mathcal{C}}$, and $\ddot{\mathcal{E}}$, this new justification is not included in the label of $\mathcal{A}$, as it is subsumed by the environment $\{\ddot{\mathcal{B}}, \ddot{\mathcal{C}}\}$.

The ATMS treats the fact `false` specially. A set of assumptions is inconsistent (*nogood*) if we can derive



$\ddot{\mathcal{B}}$
$\ddot{\mathcal{C}}$
$\mathcal{B}$
$\mathcal{C}$
$\mathcal{A}$
The truck of some.trip is Traveler
The itinerary of some.trip is ...
The truck.cost of some.trip is 452
$\{\{\ddot{\mathcal{B}}\}\}$
$\{\{\ddot{\mathcal{C}}\}\}$
$\{\{\ddot{\mathcal{B}}, \ddot{\mathcal{C}}\}\}$

FIGURE 4. The Proposition Structure of $\mathcal{A}$



𝒜
The truck.cost of some.trip is 452
{{ℬ, 𝒞}, {𝒞, 𝒟}}
ℬ
The truck of some.trip is Traveler
{{ℬ}}
$\ddot{ℬ}$
𝒞
The itinerary of some.trip is ...
{{𝒞}}
$\ddot{𝒞}$
𝒟
The truck of some.trip is Queen.Bee
{{𝒟}}
$\ddot{𝒟}$

FIGURE 5. The Proposition Structure of 𝒜 with a Second Justification

`false` from it. Traditionally, inference is the process of extending a set of beliefs by applying inference procedures to these beliefs. The use of the classical propositional logic illustrates this point: If I believe α and I believe $\alpha \supset \beta$, then I am entitled to believe β. If that process produces a contradiction, one of our original assumptions must itself be wrong. The ATMS relies on this principle. It marks as inconsistent all sets of assumptions from which contradictions have been derived and removes derivations based on such assumption sets from its working memory. In the KEE system, contradictions can be created not only by explicitly deriving `false`, but also by conclusion of both α and (`not` α), and by *cardinality* and *valueclass* violations. For simplicity's sake, however, in most of our examples we induce contradictions only by explicitly deriving `false`.

Using the ATMS as a foundation, we have built a context mechanism, much in the spirit of the contexts of QA4 [21] and Conniver [18]. We call each context a *world.* Worlds can be created interactively through the user interface, by the actions of the rule system, or programmatically. Figure 6 shows the KEEworlds Browser, a graphical representation of the worlds extant at any time. The browser shows a single world, `start`. When creating a world, the user can specify a parent world or worlds. The newly created world has, as a default, all the assumptions (and hence, derived facts) of its parent worlds.

A world is characterized by a set of assumptions—both the assumptions of the existence of that world and its ancestor worlds, and the assumptions of facts explicitly asserted and deleted in that world. Testing the context-relative belief in a proposition is straightforward: If the assumptions of a world are a superset of any of the environments in the label of the proposition, that proposition is believed in that world. The system treats as believed in a world not only those assumptions explicitly asserted into that world, but also any fact that has a derivation based on those assumptions. Thus, if in world Ψ we believe $\mathcal{B}$ and $\mathcal{C}$, we also believe $\mathcal{A}$, because $\mathcal{C}$ has been shown to be a consequence of $\mathcal{A}$ and $\mathcal{B}$. This belief carries over into this world even though the justification $\mathcal{B}, \mathcal{C} \vdash \mathcal{A}$ may have been made when the system was "focused" on some other world, perhaps even before Ψ was created. We do not have to rederive $\mathcal{A}$, because the ATMS preserves this derivation.

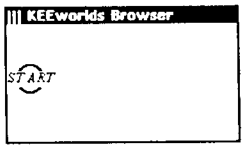


FIGURE 6. KEEworlds Browser of `Start`

```
COMPUTE.TOTAL.COST
  while
    (the truck.cost of ?t is ?vc)
    (the driver.cost of ?t is ?dc)
  believe
    (the total.cost of ?t is (+ ?dc ?vc))
COMPUTE.TRUCK.COST
  while
    (the cost.per.mile of (the truck of ?t)
     is ?x)
    (the distance of ?t is ?d)
  believe
    (the truck.cost of ?t is (* ?x ?d))
COMPUTE.DRIVER.COST
  while
    (the driver.cost.per.hour of ?t is ?w)
    (the duration of ?t is ?d)
  believe
    (the driver.cost of ?t is (* ?d ?w))
COMPUTE.DRIVER.COST.PER.HOUR
  while
    (the wage.rate of (the driver of ?t)
     is ?w)
  believe
    (the driver.cost.per.hour of ?t is ?w)
COMPUTE.DURATION
  while
    (the itineary of ?t is ?i)
    (the truck of ?t is ?v)
    (the weather of mid.continent is ?w)
  believe
    (the duration of ?t is
     (compute.duration ?i ?v ?w))
```

FIGURE 7. `Cost.Computation.Rules`

Rules and Justifications

We have spoken of the system creating justifications. In most applications, the primary source of justifications is the instantiation of *deduction rules* (although one can also explicitly add justifications). The KEE system has two kinds of rules: *deduction rules* and *action rules*. Deduction rules express the theories of a particular domain representation—truths believed in every world. Action rules create contexts and change the assumptions of particular contexts.[4]

When a deduction rule is instantiated, a justification is created. The justifications ensure that, whenever facts matching the premise of the rule are believed, the system believes the corresponding conclusions. Thus, if a deduction rule is invoked that concludes $\mathcal{X}, \mathcal{Y} \vdash \mathcal{Z}$, we do not necessarily know $\mathcal{Z}$ in any specific world. Instead, the ATMS has built a structure, the justification, that enables the system to recognize that, if we ever come to believe both $\mathcal{X}$ and $\mathcal{Y}$ in a world, we also believe $\mathcal{Z}$ in that world. Figure 7 shows the deduction rules for computing the cost of trips in our example domain.

Truth Maintenance across Worlds

The interaction of truth maintenance with worlds may seem clearer with an example. In this section we show how facts computed in one world are visible in other worlds that share the appropriate assumptions. We begin by asserting four facts in world `start`:

```
The driver of example.trip is Gray;        (E)

The truck of example.trip is Piper;        (F)

The itinerary
  of example.trip is
    '((Indianapolis nil origin)            (G)
      (Seymour computers on)
      (Thayer computers off));

The weather of Mid.Continent is snow.      (H)
```

[4] The system allows any rule to be used both for reacting to assertions (forward chaining) and for answering queries (backward chaining). Rules in KEE are written in KEE's rule language, which itself is based on the extensible query and assertion language TellAndAsk®. In our examples, the clauses of rules are either the assertion or deletion of facts, the evaluation in the underlying Lisp system of some expression, or the unification of a variable with an underlying evaluation. Unmarked facts are interpreted as assertions, and ununified Lisp expressions as Boolean tests. For the examples used in this article, the reader need only understand that a statement of the form (`the location of Cannonball is Thayer`) refers to one value of the `location` slot in unit `Cannonball` as `Thayer`, and that a statement of the form (`Cannonball is in class trucks`) means that `Cannonball` is a member of some class in the transitive closure of the subclass relation on `trucks`. TellAndAsk™ allows embedding of subexpressions (e.g., (`the transmission of (the truck of trip.1) is automatic`)) and unifies variables ("?" symbols).

™ TellAndAsk is a trademark of IntelliCorp.

Thus, world `start` includes, in its characteristic assumption set, assumptions for each of ($\mathcal{E}$)–($\mathcal{H}$). Let us call these assumptions $\breve{\mathcal{E}}$, $\breve{\mathcal{F}}$, $\breve{\mathcal{G}}$, and $\breve{\mathcal{H}}$.

When we query the system to determine the `total.cost` of `example.trip` in world `start` (using the `cost.computation.rules`), it runs the rules, deducing the fact

```
The total.cost
  of example.trip is 308.475.                ($\mathcal{I}$)
```

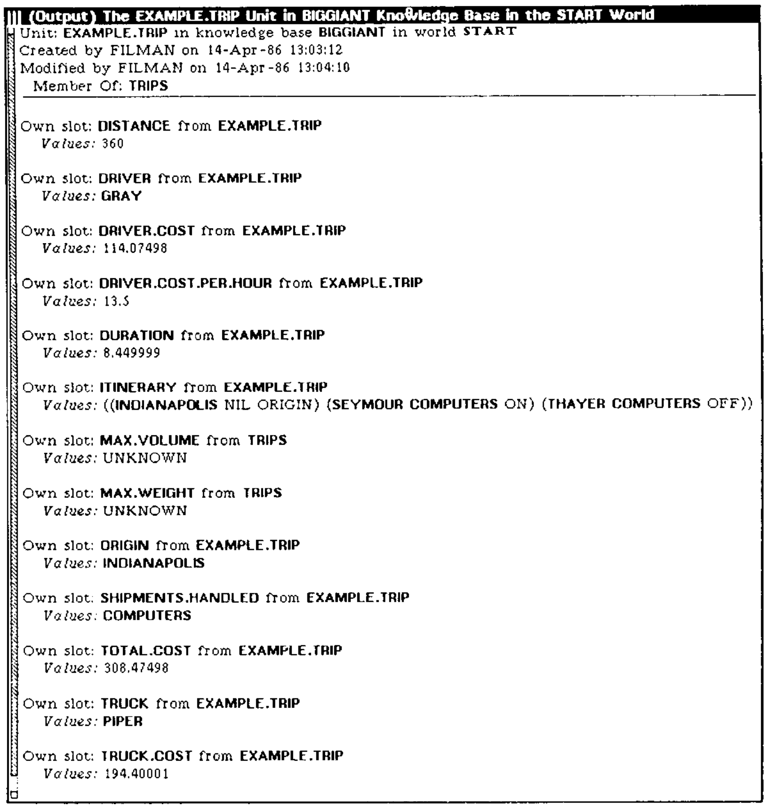
At this point, proposition $\mathcal{I}$ includes in its label the environment $\{\breve{\mathcal{E}}, \breve{\mathcal{F}}, \breve{\mathcal{G}}, \breve{\mathcal{H}}\}$. As these assumptions are a subset of the characteristic assumptions of world `start`, fact $\mathcal{I}$ is believed in `start`. Figure 8 shows a display of the unit `example.trip` relative to world `start` after this query. Since in deriving this value we derived several other intermediate values (such as the `truck.`- and `driver.costs`), the unit display also shows these values. Correspondingly, these facts have justification structures including subsets of $\{\breve{\mathcal{E}}, \breve{\mathcal{F}}, \breve{\mathcal{G}}, \breve{\mathcal{H}}\}$. (This display is a condensation (eliminating facets) of the display obtained by selecting `Display Unit` from the browser menu. In general, the user interface allows the user to browse and edit the knowledge base relative to worlds.)

Because the ATMS creates justification structures for derived facts, the justifications for beliefs are available to system and user programs. One such facility is invoked by selecting `explain` from the browser menu. Figure 9 shows the result of this selection—the *explanation graph* (proof tree) for the value of the `total.cost` of `example.trip`.

We can now create another world, `other.world`, asserting the same four facts ($\mathcal{E}$)–($\mathcal{H}$) in it. Thus, the characteristic assumptions of `other.world` include $\{\breve{\mathcal{E}}, \breve{\mathcal{F}}, \breve{\mathcal{G}}, \breve{\mathcal{H}}\}$; `other.world` sees any fact (such as $\mathcal{I}$) that is justified by this set. Figure 10 shows the browser and facts of `other.world`. If we remove, say, assumption $\breve{\mathcal{E}}$ from the characteristic set of `other.world`, the belief in the consequences of that deduction in `other.world` is withdrawn. In Figure 11 we see the beliefs of `other.world` after we have retracted fact $\mathcal{E}$ from `other.world`. The beliefs dependent on the driver being `Gray` (such as the `driver.cost` and `total.cost` of `example.trip`) are no longer present. Beliefs that do not depend on the driver, however, (such as the `truck.cost` of `example.trip`) are still there.

Contradictory Worlds

If the ATMS has been given a derivation for `false` that is justified by assumptions believed in a particular world, then that world is contradictory and is considered `nogood`. Figure 12 shows the browser after we have asserted a second truck for `example.trip` in `other.world`. Since the maximum cardinality of the



```
(Output) The EXAMPLE.TRIP Unit in BIGGIANT Knowledge Base in the START World
Unit: EXAMPLE.TRIP in knowledge base BIGGIANT in world START
Created by FILMAN on 14-Apr-86 13:03:12
Modified by FILMAN on 14-Apr-86 13:04:10
  Member Of: TRIPS

Own slot: DISTANCE from EXAMPLE.TRIP
   Values: 360

Own slot: DRIVER from EXAMPLE.TRIP
   Values: GRAY

Own slot: DRIVER.COST from EXAMPLE.TRIP
   Values: 114.07498

Own slot: DRIVER.COST.PER.HOUR from EXAMPLE.TRIP
   Values: 13.5

Own slot: DURATION from EXAMPLE.TRIP
   Values: 8.449999

Own slot: ITINERARY from EXAMPLE.TRIP
   Values: ((INDIANAPOLIS NIL ORIGIN) (SEYMOUR COMPUTERS ON) (THAYER COMPUTERS OFF))

Own slot: MAX.VOLUME from TRIPS
   Values: UNKNOWN

Own slot: MAX.WEIGHT from TRIPS
   Values: UNKNOWN

Own slot: ORIGIN from EXAMPLE.TRIP
   Values: INDIANAPOLIS

Own slot: SHIPMENTS.HANDLED from EXAMPLE.TRIP
   Values: COMPUTERS

Own slot: TOTAL.COST from EXAMPLE.TRIP
   Values: 308.47498

Own slot: TRUCK from EXAMPLE.TRIP
   Values: PIPER

Own slot: TRUCK.COST from EXAMPLE.TRIP
   Values: 194.40001
```

FIGURE 8. `Example.Trip` **in** `Start` **after Query**

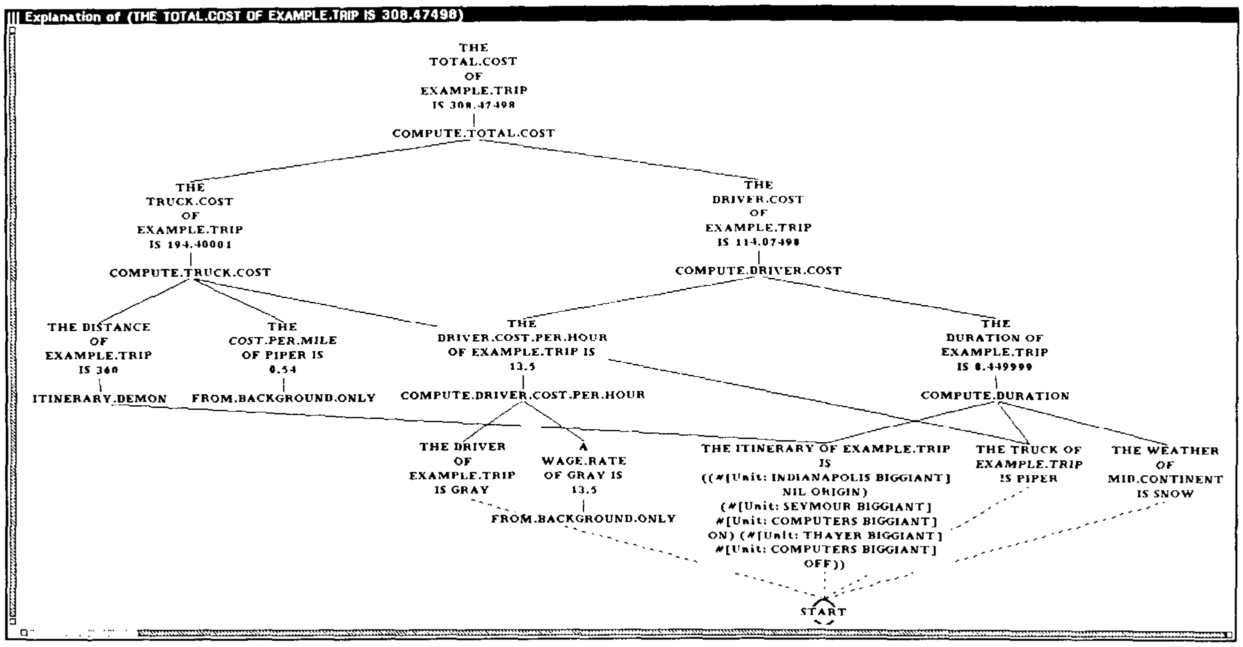
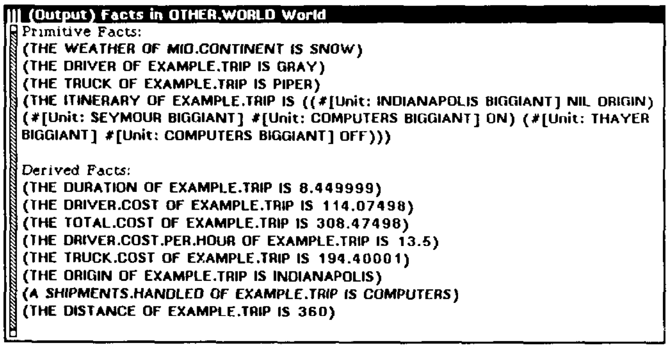

Explanation of (THE TOTAL.COST OF EXAMPLE.TRIP IS 308.47498)
THE TOTAL.COST OF EXAMPLE.TRIP IS 308.47498
COMPUTE.TOTAL.COST
THE TRUCK.COST OF EXAMPLE.TRIP IS 194.40001
COMPUTE.TRUCK.COST
THE DRIVER.COST OF EXAMPLE.TRIP IS 114.07498
COMPUTE.DRIVER.COST
THE DISTANCE OF EXAMPLE.TRIP IS 360
ITINERARY.DEMON
THE COST.PER.MILE OF PIPER IS 0.54
FROM.BACKGROUND.ONLY
THE DRIVER.COST.PER.HOUR OF EXAMPLE.TRIP IS 13.5
COMPUTE.DRIVER.COST.PER.HOUR
THE DURATION OF EXAMPLE.TRIP IS 8.449999
COMPUTE.DURATION
THE DRIVER OF EXAMPLE.TRIP IS GRAY
A WAGE.RATE OF GRAY IS 13.5
FROM.BACKGROUND.ONLY
THE ITINERARY OF EXAMPLE.TRIP IS ((#[Unit: INDIANAPOLIS BIGGIANT] NIL ORIGIN) (#[Unit: SEYMOUR BIGGIANT] #[Unit: COMPUTERS BIGGIANT] ON) (#[Unit: THAYER BIGGIANT] #[Unit: COMPUTERS BIGGIANT] OFF))
THE TRUCK OF EXAMPLE.TRIP IS PIPER
THE WEATHER OF MID.CONTINENT IS SNOW
START

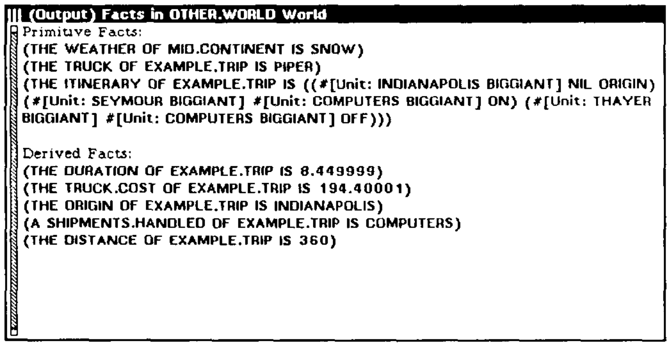
FIGURE 9. Explanation of the `Total.Cost` of `Example.Trip`



```
(Output) Facts in OTHER.WORLD World
Primitive Facts:
(THE WEATHER OF MID.CONTINENT IS SNOW)
(THE DRIVER OF EXAMPLE.TRIP IS GRAY)
(THE TRUCK OF EXAMPLE.TRIP IS PIPER)
(THE ITINERARY OF EXAMPLE.TRIP IS ((#[Unit: INDIANAPOLIS BIGGIANT] NIL ORIGIN)
(#[Unit: SEYMOUR BIGGIANT] #[Unit: COMPUTERS BIGGIANT] ON) (#[Unit: THAYER
BIGGIANT] #[Unit: COMPUTERS BIGGIANT] OFF)))

Derived Facts:
(THE DURATION OF EXAMPLE.TRIP IS 8.449999)
(THE DRIVER.COST OF EXAMPLE.TRIP IS 114.07498)
(THE TOTAL.COST OF EXAMPLE.TRIP IS 308.47498)
(THE DRIVER.COST.PER.HOUR OF EXAMPLE.TRIP IS 13.5)
(THE TRUCK.COST OF EXAMPLE.TRIP IS 194.40001)
(THE ORIGIN OF EXAMPLE.TRIP IS INDIANAPOLIS)
(A SHIPMENTS.HANDLED OF EXAMPLE.TRIP IS COMPUTERS)
(THE DISTANCE OF EXAMPLE.TRIP IS 360)
```

FIGURE 10. The Beliefs of `Other.World`



```
(Output) Facts in OTHER.WORLD World
Primitive Facts:
(THE WEATHER OF MID.CONTINENT IS SNOW)
(THE TRUCK OF EXAMPLE.TRIP IS PIPER)
(THE ITINERARY OF EXAMPLE.TRIP IS ((#[Unit: INDIANAPOLIS BIGGIANT] NIL ORIGIN)
(#[Unit: SEYMOUR BIGGIANT] #[Unit: COMPUTERS BIGGIANT] ON) (#[Unit: THAYER
BIGGIANT] #[Unit: COMPUTERS BIGGIANT] OFF)))

Derived Facts:
(THE DURATION OF EXAMPLE.TRIP IS 8.449999)
(THE TRUCK.COST OF EXAMPLE.TRIP IS 194.40001)
(THE ORIGIN OF EXAMPLE.TRIP IS INDIANAPOLIS)
(A SHIPMENTS.HANDLED OF EXAMPLE.TRIP IS COMPUTERS)
(THE DISTANCE OF EXAMPLE.TRIP IS 360)
```

FIGURE 11. The Beliefs of `Other.World` after Retracting the Driver

`truck` slot is 1 and we have asserted two different trucks for `example.trip`, we have a contradiction, and `other.world` is nogood. Nogood worlds appear in the browser with solid boxes in their centers.

When the ATMS detects a particular set of assumptions is mutually inconsistent, it propagates that information throughout the justification structure. This can result in some worlds becoming nogood. The rule system ignores nogood worlds in choosing rules to apply. We can exploit this behavior with rules that produce contradictions (e.g., by deducing `false`) in the worlds that violate domain constraints—that are logically inconsistent, undesirable, or just plain unlikely to occur in the modeled domain. Thus, deductions of contradictions can be used as a tool for controlling the reasoning process.

Constraints

We express the domain-specific constraints as deduction rules whose conclusion is `false`. For example, the dispatcher errs in assigning driver *d* to truck *v*, when *d* and *v* are in different cities. We express this constraint with the following rule:

```
DRIVER.AND.TRUCK.
    MUST.BE.IN.SAME.CITY
  while
    (the location
      of (the driver of ?t) is ?dl)
    (the location
      of (the truck of ?t) is ?vl)
    (not (equal ?dl ?vl))
  believe
    false.
```

FIGURE 12. Browser with `Other.World` Nogood

If we run this rule in a world where `Gray` (located in `Gary`) and `Queen.Bee` (located in `Indianapolis`) are the driver and truck, respectively, of `example.trip`, we deduce `false`, making the world nogood. The examples in this article were run with about a dozen constraint rules.[5]

WORLD STRUCTURES

The dispatcher is faced with a difficult problem: satisfying an irregular set of constraints while working in a large combinatorial space. This problem has aspects of "interpretation construction" (i.e., the assignment of values to a few variables) in the selection of drivers, trucks, and itineraries for trips. Itineraries are themselves complex objects, however, not amenable to simple optimization. In the next three sections, we describe a series of tools for the dispatcher: first, the *dispatcher's advice taker*, a manual approach that illustrates the use of the system to record the dispatcher's decisions and check them for constraint violations; second, a *dispatcher's apprentice* that, using the rule system, demonstrates a division of work—giving the dispatcher the hard problem of determining itineraries, and allowing the system to complete the more mechanical details of truck and driver assignments; and, third, a *dispatcher's replacement* that programmatically solves the entire problem. Our intent is to illustrate the interaction between problem solving, the ATMS, and the worlds system. Clearly, we are not presenting an interface for a dispatcher so much as the tools a system builder could use in constructing a problem solver for the dispatcher.

Problem solving is typically an exploratory, incremental process. That is, one starts with a set of beliefs about the world and recursively considers alternative choices that modify those beliefs. Usually, the modifications to a set of beliefs are incremental: By and large, we retain most of the original assumptions of the initial state, adding or deleting only a few at each step. Thus, the dispatcher who starts with the problem of completing an empty trip may choose among trucks, drivers, and itineraries to get to the next problem state; once in that state, the dispatcher may modify the itinerary or focus on an earlier point in the problem-solving process. KEEworlds allows us to reflect the structure of the search space in the structure of a worlds graph. That is, one can model alternatives or changes to a particular world by creating *child* worlds. By default, these worlds inherit the assumptions (and, therefore, the derived facts) of their parents. The user, however, is also allowed to change (add and delete) assumptions in the children (often in the creation process). Thus, we might model the action in world Ψ_p of sending `Queen.Bee` from `Indianapolis` to `New.Harmony` by creating Ψ_c, a child world of Ψ_p; and, in Ψ_c, changing `Queen.Bee`'s location to `New.Harmony`. If (as an assumption) the driver of `Queen.Bee` in Ψ_p is `Green`, `Queen.Bee`'s driver will still be `Green` in Ψ_c.

Similarly, a common problem-solving tactic is to break a problem into subproblems, solve the subproblems independently, and finally merge the subproblem solutions (if compatible) into a global solution. We model this structure by placing the original problem in a world, Ψ, and then creating children worlds, $\Psi_{c_1}, \ldots, \Psi_{c_n}$, each of which encodes one of the subproblems. When we have a set of descendant worlds, $\Psi_{d_1}, \ldots, \Psi_{d_m}$, that solve the subproblems, we try to merge them into a solution world. In the dispatcher's advice taker, we model the solution of the entire problem of schedule creation by breaking the problem into the tasks of defining a trip for each itinerary in its own separate world, and then merging these worlds (building a child world with these worlds as parents) when all tasks have been solved. This merge can fail even though each subproblem solution is itself consistent. For example, two itineraries can in themselves be consistent, but together be inconsistent because they use the same driver. We model dead ends and failures as nogood worlds.

Formally, the worlds exist in a directed, acyclic graph over the "parent–child" relation. Loosely, the assumptions true in a particular world, Ψ, are those that have been explicitly added at Ψ, and those assumptions in the parents of Ψ that have not been explicitly deleted in Ψ.[6]

THE DISPATCHER'S ADVICE TAKER

The dispatcher's advice taker leaves the decisions about trip composition to the dispatcher, but checks those decisions for consistency with the constraints of the problem space. That is, the dispatcher decides who to assign to what, and the advice taker checks to see if that assignment breaks any of the dispatching rules. (Thus, the advice taker acts in the spirit of McCarthy's Advice Taker [15]—able to converse about the domain and verify assertions, but not able to make decisions.) As described above, we express constraints as deduction rules whose conclusion is `false`.

A typical interaction with the dispatcher's advice taker might go as follows: We have three itineraries, $iter_1$, $iter_2$, and $iter_3$, for which we seek to make simultaneous truck and driver assignments:

[5] For the rules, the data, and a formal statement of the problem, write to the author.

[6] This description is a simplification of the true situation, where a more elaborate conflict-resolution strategy is used when a particular assumption has been both added and deleted in different ancestors. For the details of the conflict-resolution strategy, the reader is referred to Morris and Nado's article [19].

```
iter₁ = ((Indianapolis nil origin)
        (Seymour computers on)
        (Thayer computers off));
iter₂ = ((Gary nil origin)
        (La.Harpe toys on)
        (Viola carpet on)
        (Oregon toys off)
        (Cook carpet off));
iter₃ = ((Indianapolis nil origin)
        (Kokomo refrigerators on)
        (Warsaw refrigerators off)
        (Roselawn bicycles on)
        (Gary typewriters on)
        (Attica typewriters off)
        (Bloomington bicycles off)).
```

We create a world, `origin`, and describe the `weather` of `Mid.Continent` as `fair` in that world. We then create three child worlds of `origin`—worlds `one`, `two`, and `three`—making trips `trip.1`, `trip.2`, and `trip.3` have itineraries of *iter*$_1$, *iter*$_2$, and *iter*$_3$, respectively in those worlds (Figure 13). Effectively, we have broken the problem of finding compatible drivers and trucks for these trips into the subproblems of finding a driver and truck for each trip. We represent partially completed solutions to these problems in worlds. When each subproblem is completely solved, we merge the solutions to see if they are mutually consistent.

We try assigning `Gray` to `trip.1` by creating `Gray/one`, a child world of `one`, asserting "`the driver of trip.1 is Gray`" in that world and running the constraint rules in the new world. These rules render that world nogood. Examination of the explanation reveals that `Gray`, based in `Gary`, is unsuitable as a driver for a trip that starts in `Indianapolis` (Figure 14). Abandoning that world, we build `Green/one` and `Queen.Bee/Green/one`, where `Green` drives `Queen.Bee` on `trip.1`. Running the constraint rules in these worlds shows them free of contradictions (Figure 15). We continue in a similar fashion, finding a place for `Gray` driving `Piper` on `trip.2`, and for `White` driving `Queen.Bee` on `trip.3`. In Figure 16 we see that an attempt to merge the three leaf worlds has failed because we have assigned the same truck (`Queen.Bee`) to two different trips. We correct this with a new truck for `trip.1`, leading to a successful merge (Figure 17). This world cumulates the facts of its parents to form a solution.

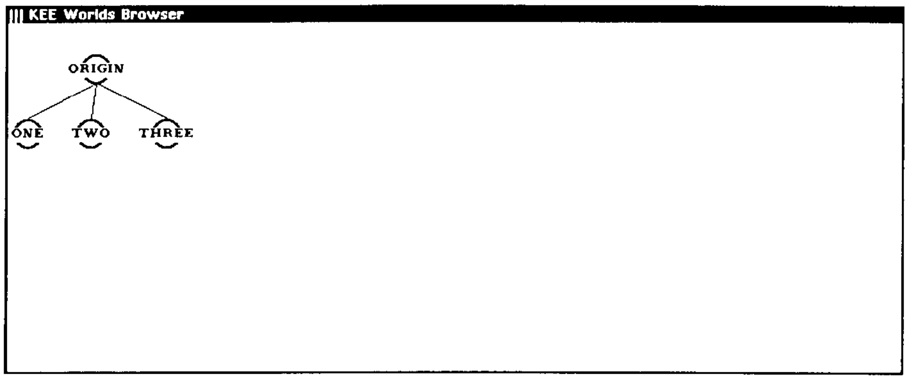


FIGURE 13. Worlds `One`, `Two`, and `Three`

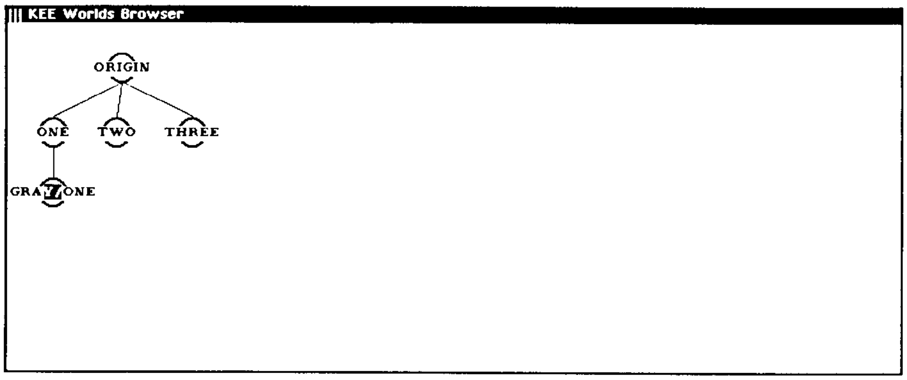


FIGURE 14. `Gray` Should Not Drive `Trip.1`

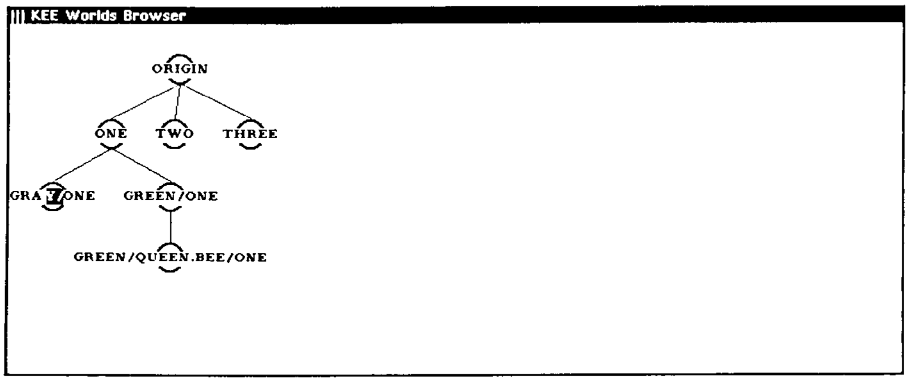


FIGURE 15. After Assignment to `Trip.1`

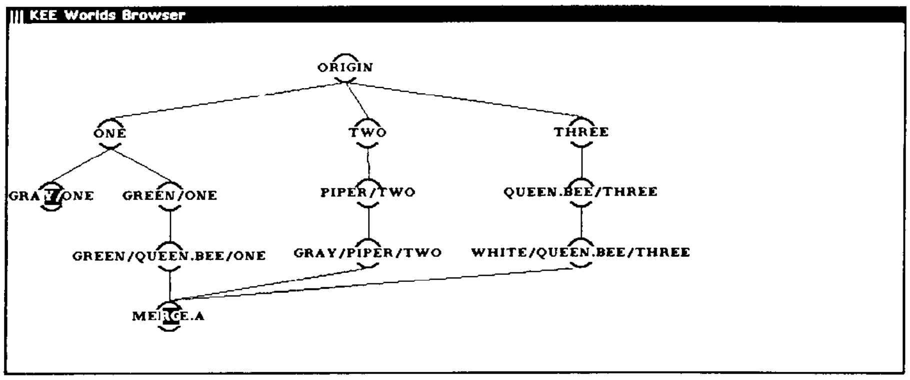


FIGURE 16. Failure of `Merge.a`

Modifying the Knowledge Base

Because we are in a dynamic, symbolic, and interactive environment, it is straightforward to modify the representation structures to reflect new concepts and constraints. If, for example, we wish to introduce the idea that (1) trucks have transmissions that are typically manual, (2) Piper has an automatic transmission, and (3) Driver Gray refuses to drive any truck with an automatic transmission, we could

(1) create member slot `transmission` in class unit `trucks`, giving it value `manual`;

(2) assert `(the transmission of Piper is automatic)`; and

(3) create another member of the class of constraint rules whose external form is as follows:

```
while
  (the driver of ?t is Gray)
  (the transmission of
   (the truck of ?t) is automatic)
believe
  false.
```

Running the constraint rules now makes world `merge.b` nogood.

This description of the dispatcher's advice taker is interesting not as an interface one would actually want to provide to a working dispatcher, but because it illustrates parent and child relationships between worlds, and shows the use of the world system to reflect problem partitioning and recombination. Worlds express problem-solving state; that state can be used for things such as segmenting knowledge, checkpointing changes, and preserving search state at the discretion of the system developer.

THE DISPATCHER'S ASSISTANT AND THE RULE SYSTEM

Most problems require more action on the part of the system than simple state preservation and constraint checking. That is, we usually want the computer to actually *solve* something, not merely represent it. KEE provides several different mechanisms for problem solving, such as active values (daemons), methods (object-oriented programming), and conventional programming. One such mechanism is the rule system. We

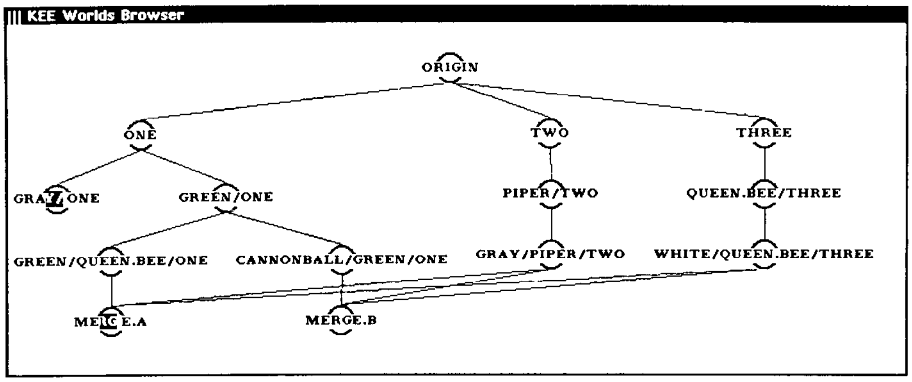


FIGURE 17. Successful Merge

have already seen examples of deduction rules—declarative expressions of universal truths. Deduction rules create justifications for facts; the ATMS ensures that worlds find those facts that are true in them. *Action rules* cause the system to *change* its view of the state of some world. That is, like the programmatic assignment of a value to a variable, action rules change the state of the system.[7]

The two kinds of action rules are *same-world action rules* and *new-world action rules*. As the name suggests, same-world rules run in a particular world; they make changes to (additions to and deletions from) the beliefs (characteristic assumptions) of that world. For a same-world rule to run in a world, all the premises of the rule must be true in that world. New-world rules create a new world. This world can have multiple parents. That is, the rule system searches for a set of compatible parent worlds where the facts have been added that satisfy the premises of the rule. For each such set, it builds a world that has that set as its parents. That new world inherits all the assumptions (and therefore deductions) of its parents. The changes implied by the action part of the rule then take place in the new world. Syntactically, new-world rules include the keyword `in.new.world` (or `in.new.and.world`) to indicate new-world creation. We distinguish action rules by the use of the keywords **`if`** and **`then`**, in place of the deduction rule's **`while`** and **`believe`**.

We trust the conclusions of the ATMS because the ATMS implements a simple monotonic propositional logic, and theorems about that logic assure us that subsumption and elimination of contradictory derivations are valid operations. To preserve this clean semantics, the premises of deduction rules must be either facts known to the ATMS or state-independent Lisp computations. Action rules express programmatic change, and we make no pretense of expressing a declarative semantics of programs. We can therefore be more liberal about premises for action rules, allowing operators such as THNOT, quantified subexpressions, mixed forward and backward chaining, and reference to non-ATMS-facts (such as class membership and member slots).

The ability of a new-world action rule to gather clauses from different worlds in creating a new child world can make it difficult to develop particular world topologies. It is often the case that the children of a world are meant as mutually exclusive alternatives. Such worlds (and their descendants) should never merge. To simplify expressing this idea, the system provides *exclusion sets*: sets of incompatible child worlds. Each world can have one or more exclusion sets. Two worlds in an exclusion set are treated as mutually contradictory and cannot share descendants. Exclusion sets can be created through the user interface, programmatically, or by the rule system. As a default, if world Ψ_c, a child of world Ψ_p, was created by the rule system, then Ψ_c is in Ψ_p's default exclusion set. It will be excluded from the other worlds created by running rules on Ψ_p. Rules specified with the keyword `in.new.and.world` create worlds that are not in any exclusion set. Exclusion sets are displayed on the browser with the symbol ⊠.

The Dispatcher's Assistant

In this section we illustrate the dispatcher's assistant, a five-action-rule system that, given a set of itineraries, determines a compatible and legal set of driver and truck assignments for those itineraries. To run the assistant, we start by creating a world, `begin`, and, for each itinerary, making it the itinerary of a trip in `begin`. We make a list of these trips, the `trip.list` of `assistant`, in `begin`. That is, in the world `begin`, we assert the following facts:

[7] The deduction/action rule division roughly corresponds to the traditional dichotomy in AI between declarative and procedural knowledge [28]. Often in AI the use of rules as facts is jumbled with their procedural interpretation. Deduction rules express universal truths and can be understood independently of their context. Action rules are, in many ways, like the procedures of a program. Independently understanding their behavior can be as difficult as understanding the import of an isolated program statement, separated from its programming context. Recognizing this difference between these two varieties of rules clarifies the issue of why rule-based systems can seem both straightforward to understand and more complicated than ordinary programming languages.

```
The weather of Mid.Continent is fair
  The itinerary of trip.1 is
    ((Indianapolis nil origin)
     (Seymour computers on)
     (Thayer computers off))
  The itinerary of trip.2 is
    ((Gary nil origin)
     (La.Harpe toys on)
     (Viola carpet on)
     (Oregon toys off)
     (Cook carpet off))
  The itinerary of trip.3 is
    ((Indianapolis nil origin)
     (Kokomo refrigerators on)
     (Warsaw refrigerators off)
     (Roselawn bicycles on)
     (Gary typewriters on)
     (Attica typewriters off)
     (Bloomington bicycles off))
  The itinerary of trip.4 is
    ((Gary nil origin)
     (Oregon books on)
     (Cook newsprint on)
     (Indianapolis newsprint off)
     (Mitchell books off))
The trip.list of assistant is
  (trip.1 trip.2 trip.3 trip.4).
```

Understanding the behavior of the assistant requires understanding the scheduling algorithm of the rule system. A rule system cycles through a three-step process of (1) determining which instantiations of rules are eligible to fire, (2) selecting a particular instantiated rule to fire, and (3) taking the actions required by that firing. Collectively, the set of instantiated rules that are eligible to fire at any cycle is that cycle's *conflict set*. A rule system's *conflict-resolution algorithm* decides which element of the conflict set fires. The conflict-resolution algorithm of the rule system is based on an agenda. When an element of the conflict set is discovered, it is added to the agenda; at each step, one of the rule instantiations on the agenda is selected for firing. The rule system's default agenda mechanism divides enabled rule instantiations into three classes: deduction rules, same-world action rules, and new-world action rules. It fires all the rules in the earlier classes before any in the later; rule instantiations in each class are kept on a stack. Thus, the default rule system behavior implements depth-first search: It tries to expand the consequences of the latest discovery first; if that fails, the system focuses on earlier situations and tries their alternatives. The rule system provides agenda functions for backward chaining using breadth-first and best-first searches, and forward chaining using combinations of rule priorities and premise complexity. Users can write their own agenda mechanisms to implement strategies such as blackboards [20].

Figure 18 shows the additional rules for the dispatcher's assistant. The system consists of five action rules in addition to the constraint rules discussed earlier. The assistant keeps its local search state on the `candidate.trucks`, `candidate.drivers`, `trip.list`, `pending.trip`, and `problem` slots of the `assistant` unit. At any point in the search, the `candidate.trucks` and `candidate.drivers` slots contain the available but not-yet-assigned trucks and drivers, and the `trip.list` slot contains a list of the trips that have not yet been filled. We denote a trip that has had a truck assigned but does not yet have a driver as a `pending.trip`. We mark the `problem` of the `assistant` as `solved` when all trips have trucks and drivers in a consistent world.

We invoke the rule system on these rules (and, of course, the constraint rules), focusing its attention on the world `begin`. This causes the following behavior:

(1) None of the constraint rules matches the data, but the first two same-world action rules (`make.candidate.trucks` and `make.candidate.drivers`) fire repeatedly, accumulating all trucks and drivers as values of the `candidate.trucks` and `candidate.drivers` slots of `assistant` in world `begin`. Because all same-world rules are run before any new-world rules, all the trucks and drivers are noticed before any assignment of a truck or driver to a trip.

(2) The new-world action rules then come into effect. `Assign.truck` selects the first trip in the `trip.list` of the current world, finds a candidate truck, and, in a new world, (1) makes that truck the truck of the trip, (2) marks that trip as the trip that is "pending" a driver, and (3) removes that truck from the set of candidate trucks. (Thus, this algorithm implicitly enforces the constraint that a truck cannot be used in more than one trip.)

(3) The constraint rules then get their turn. If they fail to make this world nogood (fail to find a contradiction), `assign.driver` continues by finding the unique pending trip, the `trip.list`, and a candidate driver, and, once again in a new world, by (1) making that driver be the driver of the trip, (2) removing that driver from the candidate drivers, (3) resetting the pending trip, and (4) setting the `trip.list` to the rest of the previous `trip.list`. Once again the constraint rules run.

In cases (2) and (3), we have selected one of several possible candidate trucks or drivers. In fact, the rule system would be perfectly happy to match these rules against every candidate truck and driver. Since the agenda is a stack, however, these other elements of the conflict set are postponed until after the consequences of the first assignment have been pursued. The system stops producing children on a particular branch when either (1) the constraints mark a world nogood, keeping the action rules from pursuing its consequences; or (2) all the possible choices at that world have been exhausted.

(4) The system, as described thus far, eventually finds all legal truck and driver assignments. The fifth rule stops the system after the first solution (literally by

clearing the agenda). After execution the browser shows a tree of worlds with root begin, each world corresponding to a decision point in the search. Figure 19 shows the browser after a run of the assistant.

PROGRAMMATIC SOLUTION: THE DISPATCHER'S REPLACEMENT

Solving the entire dispatching problem (creating itineraries, and assigning compatible trucks and drivers to them) is considerably more difficult than the previous task. For this task we turn to a programmatic solution: a Lisp program, running in the KEE system environment, that uses KEE's representation structures and invokes KEE system functionality as needed. We want a program that finds a feasible solution—one that satisfies the constraints. Since this is designed as a demonstration system, we also want a program that does this quickly.

Our strategy is to create a set of trips with compatible drivers and trucks, and empty itineraries (or, more precisely, itineraries whose sole element is that they originate in the location of their trip's driver and truck). Thus, if we have four drivers and four trucks, our set might include four trips, but incompatibilities between the available drivers and trucks might limit us to a smaller set. (E.g., if all the trucks are in one city and all the drivers in another, our maximal compatible set is empty.) We then consider each shipment in turn, looking for the best way to extend the itinerary of some trip to include it. If we are unable to find a way of extending some itinerary within the problem constraints, we start the itinerary extension process over with a (randomly) different ordering of shipments. (If several such tries all fail, we look for a different set of compatible trucks and drivers, and repeat the entire process.)[8] Our algorithm is extremely heuristic—we have sacrificed

[8] Our algorithm thus combines elements of generate-and-test and depth-first search. This is in contrast to a pure depth-first search, where the failure to place a shipment suggests trying a different alternative for the previous shipment. In our algorithm a successful placement of a shipment is not revoked unless we are trying an entirely different solution to the problem. We chose this combination in the belief that there are likely to be many solutions in the search space, but that many parts of the space lack any solutions. Hence, we want a search strategy that repeatedly samples a narrow radius over a wide area, rather than one that does a concentrated search in one place. That is, if a descent from a particular spot does not work out, it is better to try something completely different than to look too long in the same neighborhood.

```
MAKE.CANDIDATE.TRUCKS  Collect possible trucks as candidate.trucks of assistant.
  if
    (?v is in class trucks)
  then
    (a candidate.truck of assistant is ?v)
MAKE.CANDIDATE.DRIVERS  Collect possible drivers as candidate.drivers.
  if
    (?d is in class drivers)
  then
    (a candidate.driver of assistant is ?d)
ASSIGN.TRUCK  Assign a truck to this trip.
  if
    (the trip.list of assistant is ?l)
    (equal ?first (car ?l))
    (a candidate.truck of assistant is ?v)
  then in.new.world
    (delete (a candidate.truck of assistant is ?v))
    (the truck of ?first is ?v)
    (a pending.trip of assistant is ?first)
ASSIGN.DRIVER  Assign a driver to this trip.
  if
    (the trip.list of assistant is (list.of (?first . ?rest)))
    (a pending.trip of assistant is ?first)
    (a candidate.driver of assistant is ?d)
  then in.new.world
    (delete (the trip.list of assistant is (list.of (?first . ?rest))))
    (the trip.list of assistant is ?rest)
    (delete (a candidate.driver of assistant is ?d))
    (the driver of ?first is ?d)
    (delete (a pending.trip of assistant is ?first))
STOP.ASSISTANT  Stop when you've got a solution.
  if
    (the trip.list of assistant is NIL)
  then
    (the problem of assistant is solved)
    (lisp (stop.forward.chaining))
```

FIGURE 18. The Dispatcher's Assistant Rules

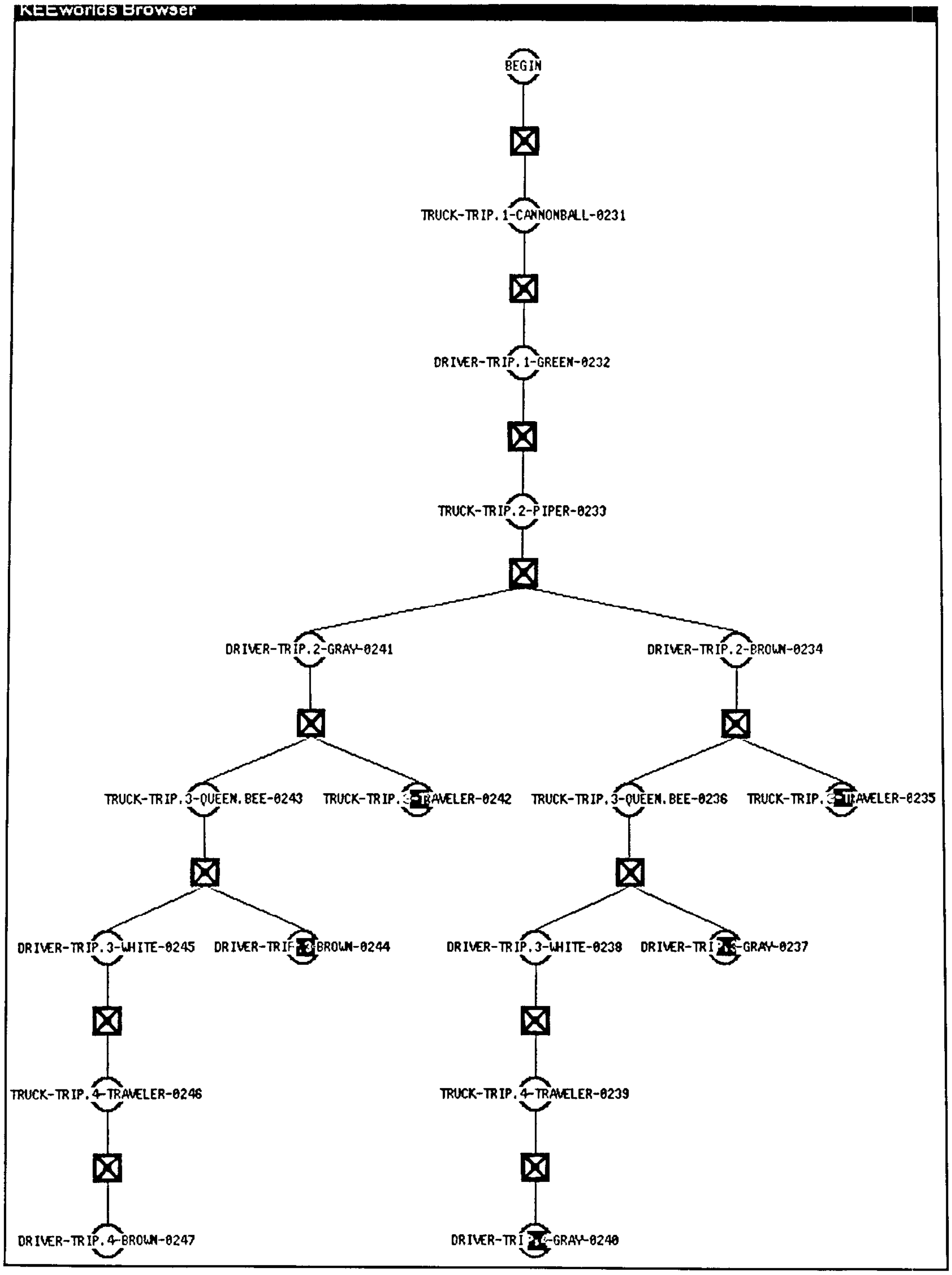


FIGURE 19. Worlds after the Dispatcher's Assistant

optimality and completeness for simplicity and speed.

We use worlds to preserve the state of our search. That is, each time the search process makes a choice, it creates a world embodying the effect of that choice. (If a world with the same information already exists, we reuse it.) Since many choices violate the constraint rules, we run the constraint rules immediately after world creation. If they deduce that the just-created world is nogood, we consider other alternatives. We consider which rule produced this nogood in deciding which other alternatives to consider. Worlds are data structures for the program; the program explicitly calls the system functions that create worlds, create exclusion sets, run the rule system, and examine the justifications of facts. Figure 20 shows the browser after running the dispatcher's replacement.

We wrote a program that searches for a feasible solution. It is natural to ask why we did not produce a more optimal algorithm, and why we have not employed the algorithmic methods of operations research, such as linear and dynamic programming, to reach this goal. Optimizing algorithms are usually considerably more computationally expensive than simply finding feasible solutions, especially in irregular domains. Our goal was not to explore the space of numerical optimization. Of course, if a close-to-optimal schedule were necessary, one could apply optimization methods, such as exchanging components of solutions and hill-climbing itinerary variations, to the results of one or several runs of the main program. Operations-research methods require restating the problem in formal, mathematical terms; make assumptions about the nature of the underlying space, for example, linearity and convexity; and are most suited to modifying an existing solution, rather than creating one from scratch. We allow any arbitrary constraints on valid solutions (e.g., "you can't ship goats and cabbage at the same time, unless you're also shipping farmers"). Our symbolic approach leaves both the problem and its solution in a form that is comprehensible to the nonexpert. It is a trivial matter, for example, for the user to define a new concept and integrate constraints that use that concept (as we showed with the transmission example). The symbolic, model-based approach makes the computational transformations of the system accessible to nonwizards. A fertile topic for research is the integration of mathematical optimization algorithms with symbolic problem expressions.

DISCUSSION

In this small example, we have been able to illustrate only a few of the potential uses of truth maintenance and the worlds system. The ATMS is a tool for search. Its primary attributes are that it preserves deductions across environments and that it retains the justifications for deductions. The first of these enables reducing

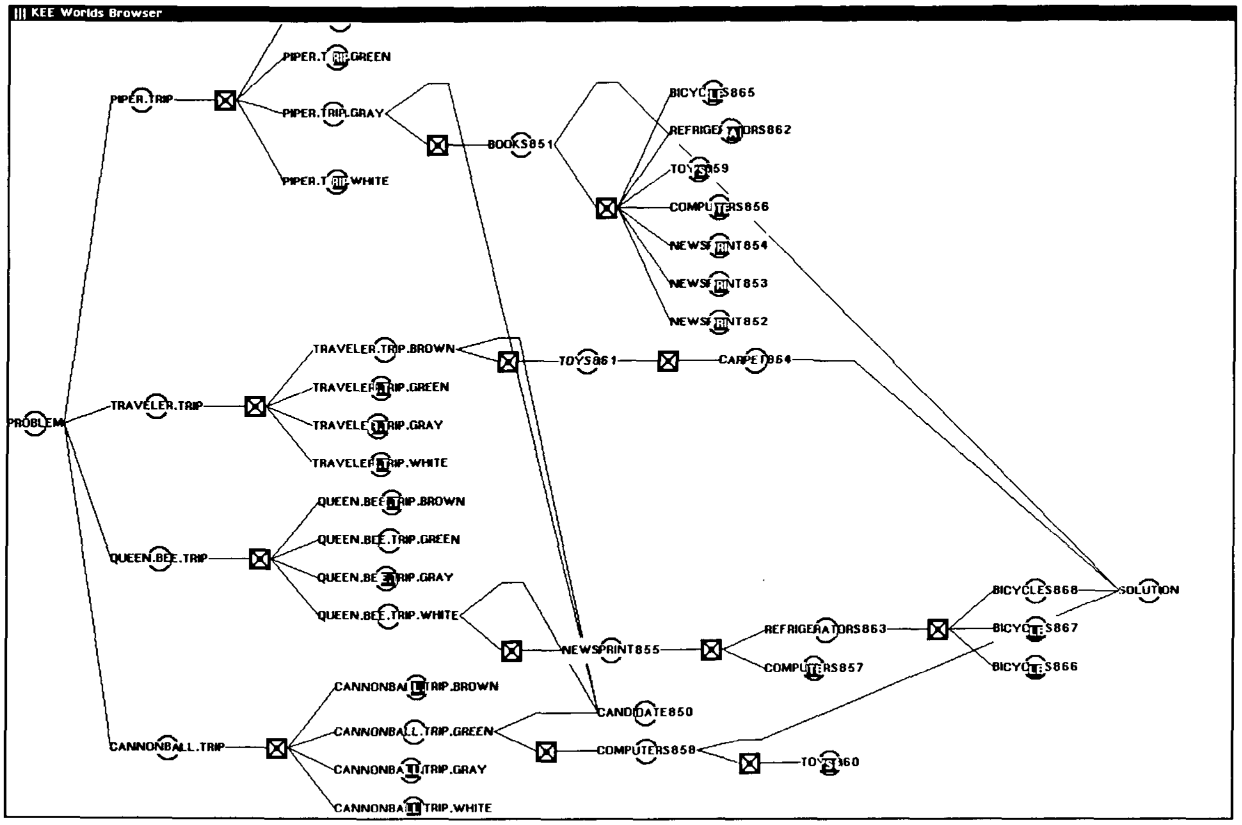


FIGURE 20. Browser after Running the Dispatcher's Replacement

search—facts do not need to be rederived, even if the system is pursuing several different alternatives simultaneously; constraints are easily expressed and automatically propagated throughout the knowledge base. The ATMS particularly lends itself to situations where the same conclusion or search state would be otherwise repeatedly rediscovered. The second allows the use of the justifications for processes such as user explanation (as in Figure 9), guiding search (as used by the dispatcher's replacement), and (though not illustrated) probabilistic and evidential reasoning. Thus, because the justification structures exist, it is straightforward to do search strategies such as dependency-directed backtracking. On the other hand, the ATMS is less appropriate for situations where nonredundant information is never derived and old contexts are rarely continued—using the ATMS requires storing information that is never retrieved.

The worlds system provides both a conceptual contexting mechanism and a systematic access to the ATMS. By reifying a group of related facts into a world, the user acquires direct access to the consequences of those facts. As we have seen, worlds can be used for state preservation and checkpointing, incremental solution construction, hypothetical reasoning, and reasoning with incomplete information. A more exhaustive list would include items such as reasoning about time and events, and representing belief structures. Because the ATMS supports the world system, derivations in one context automatically propagate to all other appropriate contexts. The ATMS also allows the worlds system a straightforward implementation of the notion of merging contexts.

We hope that the concentration of this article on examples of reasoning about scheduling under constraints has not obscured our broader point—that choosing appropriate representations and tools vastly expands the set of things a computer system can conveniently process. By using an integrated, symbolic underlying representation, we can easily model a complex domain; by employing the integrated computational tools of our system, we can easily reason about different aspects of that domain. We started with the domain of "helping the dispatcher of a trucking company." We have shown one (of the many) way of representing the data and knowledge of that domain symbolically. Because the system is symbolic, we have been spared most of the intellectual burden of coding and translating between mental concepts and machine-understandable form. Because the system provides an appropriate set of representation primitives—objects, rules, daemons, messages, truth maintenance, and worlds—it was straightforward to represent the relationships in the trucking world and reason about them. Because the representation is semantically clear, we can take the same model and use it in different ways, such as a state preserver, problem solver, user-interface basis, and simulation system. We have performed model-based reasoning.

Acknowledgments. I would like to thank William Faught, Richard Fikes, Peter Friedland, Paul Morris, Robert Nado, Anne Paulson, Marilyn Stelzner, and Eric Weiner for their insightful comments on drafts of this article. The KEEworlds/ATMS system was designed and implemented by IntelliCorp's research department: Richard Fikes, Robert Filman, Phillip McBride, Paul Morris, Robert Nado, Anne Paulson, Richard Treitel, and Martin Yonke.

REFERENCES

1. Brachman, R.J., and Schmolze, J.G. An overview of the KL-ONE knowledge representation system. *Cognitive Sci. 9*, 2 (Apr. 1985), 171–216.
2. Brownston, L., Farrell, R., Kant, E., and Martin, N. *Programming Expert Systems in OPS5.* Addison-Wesley, Reading, Mass., 1985.
3. de Kleer, J. An assumption-based truth maintenance system. *Artif. Intell. 28*, 2 (Jan. 1986), 127–162.
4. de Kleer, J., Doyle, J., Steele, G.L., and Sussman, G.J. Explicit control of reasoning. In *Artificial Intelligence: An MIT Perspective*, P.H. Winston and R.H. Brown, Eds. MIT Press, Cambridge, Mass., 1979, pp. 93–116.
5. Doyle, J. A truth maintenance system. *Artif. Intell. 12*, 3 (1979), 231–272.
6. Fikes, R., and Kehler, T. The role of frame-based representation in reasoning. *Commun. ACM 28*, 9 (Sept. 1985), 904–920.
7. Fikes, R., and Nilsson, N.J. STRIPS: A new approach to the application of theorem proving to problem solving. *Artif. Intell. 2*, 3–4 (1971), 189–208.
8. Fikes, R., Nado, R., Filman, R., McBride, P., Morris, P., Paulson, A., Treitel, R., and Yonke, M. *OPUS: A New Generation Knowledge Engineering Environment. Phase 1 Final Report.* IntelliCorp, Mountain View, Calif., 1987.
9. Goldberg, A., and Robson, D. *SMALLTALK-80: The Language and Its Implementation.* Addison-Wesley, Reading, Mass., 1983.
10. Hayes-Roth, F., and London, P. Software speeds expert systems. *Syst. Softw. 71* (Aug. 1985), 71–75.
11. Hewitt, C. Description and theoretical analysis (using schemata) of PLANNER: A language for proving theorems and manipulating models in a robot. Doct. dissertation, Mathematics Dept., MIT, Cambridge, Mass., 1971.
12. London, P. Dependency networks as representation for modelling in general problem solvers. Tech. Rep. 698, Dept. of Computer Science, Univ. of Maryland, College Park, 1978.
13. Martins, J.P., and Shapiro, S.C. Reasoning in multiple belief spaces. In *Proceedings of the 8th International Joint Conference on Artificial Intelligence* (Karlsruhe, W. Germany, Aug.). International Joint Conference on Artificial Intelligence, 1983, pp. 370–372.
14. McAllester, D. Reasoning utility package user's manual. AIM 667, Artificial Intelligence Laboratory, MIT, Cambridge, Mass., 1982.
15. McCarthy, J. Programs with common sense. In *Semantic Information Processing*. M. Minsky, Ed. MIT Press, Cambridge, Mass., 1968, pp. 403–418.
16. McCarthy, J., and Hayes, P. Some philosophical problems from the standpoint of artificial intelligence. In *Machine Intelligence 4*, B. Meltzer and D. Michie, Eds. Edinburgh University Press, Edinburgh, U.K., 1969, pp. 463–502.
17. McDermott, D. Contexts and data dependencies: A synthesis. *IEEE Trans. Pattern Anal. Mach. Intell. 5*, 3 (May 1983), 237–246.
18. McDermott, D.V., and Sussman, G.J. The Conniver reference manual. Memo 259, AI Laboratory, MIT, Cambridge, Mass., revised July 1973.
19. Morris, P.H., and Nado, R.A. Representing actions with an assumption-based truth maintenance system. In *Proceedings of the 5th National Conference on Artificial Intelligence* (Philadelphia, Pa., Aug.). American Association for Artificial Intelligence, 1986. pp. 13–17.
20. Nii, P. The blackboard model of problem solving. *AI Mag. 7*, 2 (Summer 1986), 38–53.
21. Rulifson, J.F., Derksen, J.A., and Waldinger, R.J. QA4: A procedural calculus for intuitive reasoning. Tech. Note 73, Artificial Intelligence Center, Stanford Research Institute, Menlo Park, Calif., Nov. 1972.
22. Stallman, R.M., and Sussman, G.J. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artif. Intell. 9*, 2 (Oct. 1977), 135–196.

23. Stefik, M. An examination of a frame-structured representation system. In *Proceedings of the 6th International Joint Conference on Artificial Intelligence* (Tokyo, Japan). International Joint Conference on Artificial Intelligence, 1979, pp. 845–852.
24. Stefik, M., Bobrow, D.G., Mittal, S., and Conway, L. Knowledge programming in Loops: Report on an experimental course. *AI Mag. 4*, 3 (Fall 1983), pp. 3–13.
25. Symbolics Corp. *User's Guide to Symbolics Computers, Volume 4.* Symbolics Corp., Cambridge, Mass., 1985, pp. 111–146.
26. van Melle, W., Scott, A.C., Bennett, J.S., and Peairs, M.A. The EMYCIN Manual. Tech. Rep. HPP-81-16. Heuristic Programming Project, Stanford Univ., Stanford, Calif., 1981.
27. Williams, C. *ART the Advanced Resoning Tool—Conceptual Overview.* Inference Corp., Los Angeles, Calif., 1984.
28. Winograd, T. Frame representations and the declarative/procedural controversy. In *Representation and Understanding*, D.G. Bobrow and A. Collins, Eds. Academic Press, New York, 1975, pp. 185–210.

CR Categories and Subject Descriptors: D.2.6 [**Software Engineering**]: Programming Environments; D.3.2 [**Programming Languages**]: Language Classifications—*very high-level languages*; I.2.1 [**Artificial Intelligence**]: Applications and Expert Systems; I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving—*nonmonotonic reasoning and belief revision*; I.2.4 [**Artificial Intelligence**]: Knowledge Representation Formalisms and Methods—*frames and scripts*; *representation languages*; I.2.5 [**Artificial Intelligence**]: Programming Languages and Software—*expert system tools and techniques*; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods and Search

General Terms: Design, Languages

Additional Key Words and Phrases: Assumption-based truth maintenance system (ATMS), KEE, KEEworlds, knowledge-based systems, object-based representations

Author's Present Address: Robert E. Filman, IntelliCorp, 1975 El Camino Real West, Mountain View, CA 94040.